MAJOR PROJECT – II

Programming a Self-Driving Car using

Artificial Intelligence Methods

A project report submitted in partial fulfillment of the requirements

for the award of degree of

Bachelor of Technology

In

Computer Engineering

By

**Prateek Narang** **Shivam Kapoor**

(2K12/CO/091) (2K12/CO/120)

**Rohit Maan Subham Mishra**

(2K12/CO/106) (2K12/CO/131)

Under the guidance of

**Mr. Manoj Kumar**

Department of Computer Engineering



Department of Computer Engineering

Delhi Technological University

(Formerly Delhi College of Engineering)

New Delhi-110042

**CERTIFICATE**

This is to certify that the Major Project-II entitled ‘Programming a Self-Driving Car using Artificial Intelligence Methods’ submitted by Subham Mishra, Shivam Kapoor, Prateek Narang and Rohit Maan towards the partial fulfilment of the requirements of the award of Bachelor of Technology degree in Computer Engineering at Delhi Technological University is an authentic work carried out by them under my supervision and guidance.

**Mr. Manoj Kumar**

**Project Guide**

**Computer Engineering Department**

**Delhi Technological University**

**Delhi-110042**

**ACKNOWLEDGEMENT**

We would like to express our sincere gratitude to our guide and mentor Mr. Manoj Kumar, for his guidance and help extended at every stage of this project work. We are deeply indebted to his for giving us a definite direction and moral support to complete the project successfully.

We are thankful to the Department of Computer Engineering, Delhi Technological University for all their help in completion of this project.

Prateek Narang Shivam Kapoor

(2K12/CO/091) (2K12/CO/120)

Rohit Mann Subham Mishra

(2K12/CO/106) (2K12/CO/131)

**DECLARATION**

We hereby that the Major Project-II work entitled ‘Programming a Self-Driving Car using Artificial Intelligence Methods’ which is being submitted to Delhi Technological University for the award of the degree of Bachelor of Technology in the department of Computer Engineering is a bona fide report of Major Project-II carried out by us. The material contained in the report has not been submitted to any other University or Institution for the award of any degree.

Prateek Narang Shivam Kapoor

(2K12/CO/091) (2K12/CO/120)

Rohit Mann Subham Mishra

(2K12/CO/106) (2K12/CO/131)

**Contents**

CERTIFICATE 2

ACKNOWLEDGEMENT 3

DECLERATION 4

ABSTRACT 6

CHAPTER 1: INTRODUCTION 7

CHAPTER 2: ARTIFICIAL INTELLIGENCE 9

CHAPTER 3: LOCALIZATION 18

CHAPTER 4: KALMAN FILTER 22

CHAPTER 5: PARTICLE FILTER 26

CHAPTER 6: MOTION CONTROL 34

CHAPTER 7: PID CONTROLLER 45

CHAPTER 8: CONCLUSION 51

CHAPTER 9: REFERENCES 52

**ABSTRACT**

Major concerns included "The technology is ahead of the level of governance in many areas," citing "all presume to have a human being operating the vehicle". "The technology is now advancing so quickly that it is in danger of outstripping existing rules" Such major concerns have stalled the research for a great time. With promising trials and further advancement it has made driverless car or an autonomous car an area of great research and expertise.

**Problem Statement:**

To create and optimize a software implementation required for the detection of location i.e. localization, dynamic path finding from a fixed source to fixed destination in a real time scenario and also the necessary controller for the efficient control of an autonomous or the self-driving car.

**Solution:**

We will be using several artificial intelligence techniques for localization such as Kalman Filters, Histogram Filters and the heuristic approach algorithm A\* for dynamic path finding.

Chapter 1: Introduction

AI research is highly technical and specialized, and is deeply divided into subfields that often fail to communicate with each other. Some of the division is due to social and cultural factors: subfields have grown up around particular institutions and the work of individual researchers. AI research is also divided by several technical issues. Some subfields focus on the solution of specific problems. Others focus on one of several possible approaches or on the use of a particular tool or towards the accomplishment of particular applications.

In this project, we will implement a program for Google’s self-driving car by using several artificial intelligence techniques for localization such as Kalman Filters, Histogram Filters and the heuristic approach algorithm A\* for dynamic path finding.

The dynamic path finding is very useful as it will give instant path directions based on the current car location. The A\* algorithm, which is used to implement path based on the current and final location is very fast in comparison with Dijkstra and BFS. Later we implement PID controller to find the smoothest path to the destination as A\* gives only square paths.

**1.1 Motivation**

The incredibly complicated technology behind self-driving cars lets the on board computer make hundreds of calculations a second. These include how far you are from objects, current speed, behavior of other cars, and location on the globe. These super accurate readings have virtually eliminated driving errors for test cars on the road, as the only accidents so far are while human drivers have been in control.

Because self-driving cars are rarely involved in accidents, their potential to ease congestion is high. Not only that, because self-driving cars can communicate with each other, they would eliminate the need for traffic signals. By driving at a slower rate but with less stops, better coordinated traffic would lead to less congestion.

**1.2 Aim of Project**

The aim of the project is to implement A\*algorithm to find dynamic path for the car to reach destination and to implement PID controller on the path derived by the algorithm to find the smoothest path for the car to reach the destination.

**1.3 Thesis Outline**

The thesis is divided into eight chapters and appendix.

Chapter 1 is the introduction part. It describes motivation of work, aim of thesis and the structure

of thesis.

Chapter 2 presents the Artificial Intelligence in detail, which includes the various methods and tools used to implement artificial intelligence.

and their contribution upon which we have built up.

Chapter 3 discusses about Localization. It tells what Localization and how it is implemented in the system.

Chapter 5 discusses about particle filter. It tells what Particle filter and how it is implemented in the system.

Chapter 6 discusses about the planning of motion of the car.

Chapter 7 discusses PID Control

Chapter 8 discusses the result of our project.

Chapter 9 discusses the future work.

.

Chapter 2: Artificial Intelligence

**2.1 Introduction**

Artificial intelligence (AI) is the intelligence exhibited by machines. In computer science, an ideal "intelligent" machine is a flexible rational agent that perceives its environment and takes actions that maximize its chance of success at an arbitrary goal. Colloquially, the term "artificial intelligence" is likely to be applied when a machine uses cutting-edge techniques to competently perform or mimic "cognitive" functions that we intuitively associate with human minds, such as "learning" and "problem solving". The colloquial connotation, especially among the public, associates artificial intelligence with machines that are "cutting-edge" (or even "mysterious"). This subjective borderline around what constitutes "artificial intelligence" tends to shrink over time; for example, optical character recognition is no longer perceived as an exemplar of "artificial intelligence" as it is nowadays a mundane routine technology. Modern examples of AI include computers that can beat professional players at Chess and Go, and self-driving cars that navigate crowded city streets.

AI research is highly technical and specialized, and is deeply divided into subfields that often fail to communicate with each other. Some of the division is due to social and cultural factors: subfields have grown up around particular institutions and the work of individual researchers. AI research is also divided by several technical issues. Some subfields focus on the solution of specific problems. Others focus on one of several possible approaches or on the use of a particular tool or towards the accomplishment of particular applications.

The central problems (or goals) of AI research include reasoning, knowledge, planning, learning, natural language processing (communication), perception and the ability to move and manipulate objects. General intelligence is still among the field's long-term goals. Currently popular approaches include statistical methods, computational intelligence and traditional symbolic AI. There are a large number of tools used in AI, including versions of search and mathematical optimization, logic, methods based on probability and economics, and many others. The AI field is interdisciplinary, in which a number of sciences and professions converge, including computer science, mathematics, psychology, linguistics, philosophy and neuroscience, as well as other specialized fields such as artificial psychology.

**2.2 Goals**

The general problem of simulating (or creating) intelligence has been broken down into a number of specific sub-problems. These consist of particular traits or capabilities that researchers would like an intelligent system to display. The traits described below have received the most attention.

**2.2.1 Deduction, reasoning, problem solving**

Early AI researchers developed algorithms that imitated the step-by-step reasoning that humans use when they solve puzzles or make logical deductions.[41] By the late 1980s and 1990s, AI research had also developed highly successful methods for dealing with uncertain or incomplete information, employing concepts from probability and economics.

For difficult problems, most of these algorithms can require enormous computational resources – most experience a "combinatorial explosion": the amount of memory or computer time required becomes astronomical when the problem goes beyond a certain size. The search for more efficient problem-solving algorithms is a high priority for AI research.

Human beings solve most of their problems using fast, intuitive judgments rather than the conscious, step-by-step deduction that early AI research was able to model. AI has made some progress at imitating this kind of "sub-symbolic" problem solving: embodied agent approaches emphasize the importance of sensorimotor skills to higher reasoning; neural net research attempts to simulate the structures inside the brain that give rise to this skill; statistical approaches to AI mimic the probabilistic nature of the human ability to guess.

**2.2.2 Knowledge representation**

An ontology represents knowledge as a set of concepts within a domain and the relationships between those concepts.

Main articles: Knowledge representation and commonsense knowledge

Knowledge representation and knowledge engineering are central to AI research. Many of the problems machines are expected to solve will require extensive knowledge about the world. Among the things that AI needs to represent are: objects, properties, categories and relations between objects; situations, events, states and time; causes and effects; knowledge about knowledge (what we know about what other people know); and many other, less well researched domains. A representation of "what exists" is an ontology: the set of objects, relations, concepts and so on that the machine knows about. The most general are called upper ontologies, which attempt to provide a foundation for all other knowledge.

**2.2.3 Planning**

A hierarchical control system is a form of control system in which a set of devices and governing software is arranged in a hierarchy.

Main article: Automated planning and scheduling

Intelligent agents must be able to set goals and achieve them. They need a way to visualize the future (they must have a representation of the state of the world and be able to make predictions about how their actions will change it) and be able to make choices that maximize the utility (or "value") of the available choices.

In classical planning problems, the agent can assume that it is the only thing acting on the world and it can be certain what the consequences of its actions may be. However, if the agent is not the only actor, it must periodically ascertain whether the world matches its predictions and it must change its plan as this becomes necessary, requiring the agent to reason under uncertainty.

Multi-agent planning uses the cooperation and competition of many agents to achieve a given goal. Emergent behavior such as this is used by evolutionary algorithms and swarm intelligence.

**2.2.4 Learning**

Machine learning is the study of computer algorithms that improve automatically through experience and has been central to AI research since the field's inception.

Unsupervised learning is the ability to find patterns in a stream of input. Supervised learning includes both classification and numerical regression. Classification is used to determine what category something belongs in, after seeing a number of examples of things from several categories. Regression is the attempt to produce a function that describes the relationship between inputs and outputs and predicts how the outputs should change as the inputs change. In reinforcement learning the agent is rewarded for good responses and punished for bad ones. The agent uses this sequence of rewards and punishments to form a strategy for operating in its problem space. These three types of learning can be analyzed in terms of decision theory, using concepts like utility. The mathematical analysis of machine learning algorithms and their performance is a branch of theoretical computer science known as computational learning theory.

Within developmental robotics, developmental learning approaches were elaborated for lifelong cumulative acquisition of repertoires of novel skills by a robot, through autonomous self-exploration and social interaction with human teachers, and using guidance mechanisms such as active learning, maturation, motor synergies, and imitation.

**2.2.5 Natural language processing (communication)**

A parse tree represents the syntactic structure of a sentence according to some formal grammar.

Main article: Natural language processing

Natural language processing gives machines the ability to read and understand the languages that humans speak. A sufficiently powerful natural language processing system would enable natural language user interfaces and the acquisition of knowledge directly from human-written sources, such as newswire texts. Some straightforward applications of natural language processing include information retrieval (or text mining), question answering and machine translation.

A common method of processing and extracting meaning from natural language is through semantic indexing. Increases in processing speeds and the drop in the cost of data storage makes indexing large volumes of abstractions of the user's input much more efficient.

**2.2.6 Perception**

Machine perception is the ability to use input from sensors (such as cameras, microphones, tactile sensors, sonar and others more exotic) to deduce aspects of the world. Computer vision is the ability to analyze visual input. A few selected sub problems are speech recognition, facial recognition and object recognition.

**2.2.8 Motion and manipulation**

The field of robotics is closely related to AI. Intelligence is required for robots to be able to handle such tasks as object manipulation[80] and navigation, with sub-problems of localization (knowing where you are, or finding out where other things are), mapping (learning what is around you, building a map of the environment), and motion planning (figuring out how to get there) or path planning (going from one point in space to another point, which may involve compliant motion – where the robot moves while maintaining physical contact with an object).

**2.2.9 Social intelligence**

Affective computing is the study and development of systems and devices that can recognize, interpret, process, and simulate human affects. It is an interdisciplinary field spanning computer sciences, psychology, and cognitive science. While the origins of the field may be traced as far back as to early philosophical inquiries into emotion,[87] the more modern branch of computer science originated with Rosalind Picard's 1995 paper[88] on affective computing. A motivation for the research is the ability to simulate empathy. The machine should interpret the emotional state of humans and adapt its behavior to them, giving an appropriate response for those emotions.

Emotion and social skills play two roles for an intelligent agent. First, it must be able to predict the actions of others, by understanding their motives and emotional states. (This involves elements of game theory, decision theory, as well as the ability to model human emotions and the perceptual skills to detect emotions.) Also, in an effort to facilitate human-computer interaction, an intelligent machine might want to be able to display emotions—even if it does not actually experience them itself—in order to appear sensitive to the emotional dynamics of human interaction.

**2.3 Tools Used to Implement AI**

In the course of 50 years of research, AI has developed a large number of tools to solve the most difficult problems in computer science. A few of the most general of these methods are discussed below.

**2.3.1 Search and optimization**

Many problems in AI can be solved in theory by intelligently searching through many possible solutions: Reasoning can be reduced to performing a search. For example, logical proof can be viewed as searching for a path that leads from premises to conclusions, where each step is the application of an inference rule. Planning algorithms search through trees of goals and sub goals, attempting to find a path to a target goal, a process called means-ends analysis. Robotics algorithms for moving limbs and grasping objects use local searches in configuration space. Many learning algorithms use search algorithms based on optimization.

Simple exhaustive searches are rarely sufficient for most real world problems: the search space (the number of places to search) quickly grows to astronomical numbers. The result is a search that is too slow or never completes. The solution, for many problems, is to use "heuristics" or "rules of thumb" that eliminate choices that are unlikely to lead to the goal (called "pruning the search tree"). Heuristics supply the program with a "best guess" for the path on which the solution lies. Heuristics limit the search for solutions into a smaller sample size.

A very different kind of search came to prominence in the 1990s, based on the mathematical theory of optimization. For many problems, it is possible to begin the search with some form of a guess and then refine the guess incrementally until no more refinements can be made. These algorithms can be visualized as blind hill climbing: we begin the search at a random point on the landscape, and then, by jumps or steps, we keep moving our guess uphill, until we reach the top. Other optimization algorithms are simulated annealing, beam search and random optimization.

Evolutionary computation uses a form of optimization search. For example, they may begin with a population of organisms (the guesses) and then allow them to mutate and recombine, selecting only the fittest to survive each generation (refining the guesses). Forms of evolutionary computation include swarm intelligence algorithms (such as ant colony or particle swarm optimization)and evolutionary algorithms (such as genetic algorithms, gene expression programming, and genetic programming).

**2.3.2 Logic**

Logic is used for knowledge representation and problem solving, but it can be applied to other problems as well. For example, the satplan algorithm uses logic for planning and inductive logic programming is a method for learning.

Several different forms of logic are used in AI research. Propositional or sentential logic is the logic of statements which can be true or false. First-order logic also allows the use of quantifiers and predicates, and can express facts about objects, their properties, and their relations with each other. Fuzzy logic, is a version of first-order logic which allows the truth of a statement to be represented as a value between 0 and 1, rather than simply True (1) or False (0). Fuzzy systems can be used for uncertain reasoning and have been widely used in modern industrial and consumer product control systems. Subjective logic models uncertainty in a different and more explicit manner than fuzzy-logic: a given binomial opinion satisfies belief + disbelief + uncertainty = 1 within a Beta distribution. By this method, ignorance can be distinguished from probabilistic statements that an agent makes with high confidence.

Default logics, non-monotonic logics and circumscription are forms of logic designed to help with default reasoning and the qualification problem. Several extensions of logic have been designed to handle specific domains of knowledge, such as: description logics; situation calculus, event calculus and fluent calculus (for representing events and time); causal calculus; belief calculus; and modal logics.

**2.3.3 Probabilistic methods for uncertain reasoning**

Main articles: Bayesian network, Hidden Markov model, Kalman filter, Decision theory and Utility theory

Many problems in AI (in reasoning, planning, learning, perception and robotics) require the agent to operate with incomplete or uncertain information. AI researchers have devised a number of powerful tools to solve these problems using methods from probability theory and economics.

Bayesian networks are a very general tool that can be used for a large number of problems: reasoning (using the Bayesian inference algorithm), learning (using the expectation-maximization algorithm), planning (using decision networks) and perception (using dynamic Bayesian networks). Probabilistic algorithms can also be used for filtering, prediction, smoothing and finding explanations for streams of data, helping perception systems to analyze processes that occur over time (e.g., hidden Markov models or Kalman filters).

A key concept from the science of economics is "utility": a measure of how valuable something is to an intelligent agent. Precise mathematical tools have been developed that analyze how an agent can make choices and plan, using decision theory, decision analysis, and information value theory. These tools include models such as Markov decision processes, dynamic decision networks,[ game theory and mechanism design.

**2.3.4 Classifiers and statistical learning methods**

The simplest AI applications can be divided into two types: classifiers ("if shiny then diamond") and controllers ("if shiny then pick up"). Controllers do, however, also classify conditions before inferring actions, and therefore classification forms a central part of many AI systems. Classifiers are functions that use pattern matching to determine a closest match. They can be tuned according to examples, making them very attractive for use in AI. These examples are known as observations or patterns. In supervised learning, each pattern belongs to a certain predefined class. A class can be seen as a decision that has to be made. All the observations combined with their class labels are known as a data set. When a new observation is received, that observation is classified based on previous experience.

A classifier can be trained in various ways; there are many statistical and machine learning approaches. The most widely used classifiers are the neural network, kernel methods such as the support vector machine, k-nearest neighbor algorithm, Gaussian mixture model, naive Bayes classifier, and decision tree. The performance of these classifiers have been compared over a wide range of tasks. Classifier performance depends greatly on the characteristics of the data to be classified. There is no single classifier that works best on all given problems; this is also referred to as the "no free lunch" theorem. Determining a suitable classifier for a given problem is still more an art than science.

**2.3.5 Neural networks**

A neural network is an interconnected group of nodes, akin to the vast network of neurons in the human brain. The study of non-learning artificial neural networks began in the decade before the field of AI research was founded, in the work of Walter Pitts and Warren McCullough. Frank Rosenblatt invented the perceptron, a learning network with a single layer, similar to the old concept of linear regression. Early pioneers also include Alexey Grigorevich Ivakhnenko, Teuvo Kohonen, Stephen Grossberg, Kunihiko Fukushima, Christoph von der Malsburg, David Willshaw, Shun-Ichi Amari, Bernard Widrow, John Hopfield, and others.

The main categories of networks are acyclic or feedforward neural networks (where the signal passes in only one direction) and recurrent neural networks (which allow feedback and short-term memories of previous input events). Among the most popular feedforward networks are perceptrons, multi-layer perceptrons and radial basis networks. Neural networks can be applied to the problem of intelligent control (for robotics) or learning, using such techniques as Hebbian learning, GMDH or competitive learning.

Today, neural networks are often trained by the backpropagation algorithm, which had been around since 1970 as the reverse mode of automatic differentiation published by Seppo Linnainmaa, and was introduced to neural networks by Paul Werbos.

Hierarchical temporal memory is an approach that models some of the structural and algorithmic properties of the neocortex.

**2.3.6 Control theory**

Control theory is an interdisciplinary branch of engineering and mathematics that deals with the behavior of dynamical systems with inputs, and how their behavior is modified by feedback.

**2.3.7 Languages**

AI researchers have developed several specialized languages for AI research, including Lisp and Prolog.

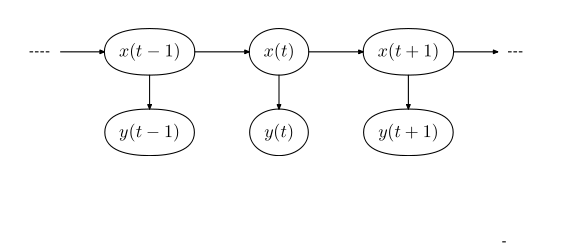
Chapter 3: Localization

**3.1 Introduction**

In order to drive the car must first know its location within its environment however it is not possible to use an instrument to measure exactly where a driverless car is. Even the best GPS sensors have a margin of error of around 5 meters (and think about what that means for a car given that the standard lane width is around 3.5 meters). Finding where a robot is when it can't measure location directly is called localization. There are several AI techniques that allow a driverless car to incorporate temporal, noisy input to generate a probabilistically sound estimate.

For localization, the underlying model is generally some form of Bayesian model similar to a hidden markov model where the state space of the the unknown "location" variables are continuous. At each time point *t* there are two variables: an unknown variable which is the location of the car, *x(t)*, and observations about the car's location based on the sensor inputs at that given time, *y(t)*. The model assumes that x(t) is generated from x(t - 1) with some unknown distribution and that y(t) is generated from x(t) with some unknown distribution.

Bayesian Model



It mainly used a particle filters algorithm, a randomized algorithm which repeatedly samples possible scenarios, to come up with a best estimate for the where it is: in the diagram above the variables *x(t)*.

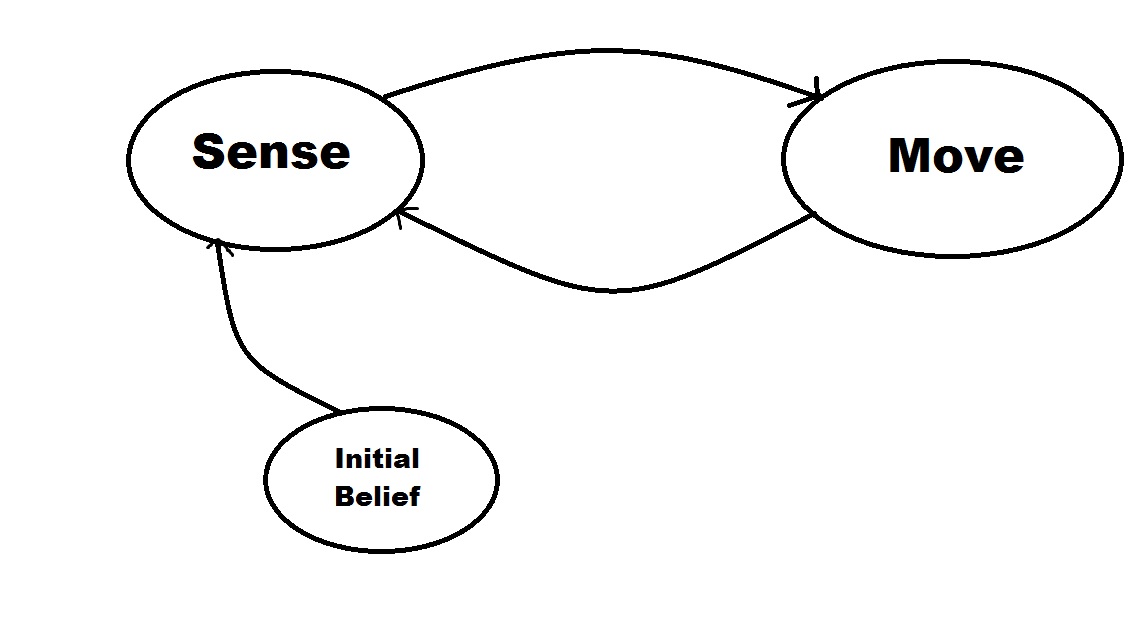
**3.2 Perception**

Once a car knows where it is, its next job is to perceive and track objects it might collide with (for example other cars or pedastrians). This problem has many similarities to localization. The sensors of other objects are noisy and we want to predict where they are. Again, the underlying model is a modification of a hidden markov model to allow for continuous variables. The picture below shows a variable representation. At each time point there are two variables: the

However, though a robot represents localizing itself and perceiving others with very similar underlying models, the two problems are solved using different algorithms. Instead of using a particle filter to estimate position, perceiving other objects is solved with kalman filters. Kalman filters make an additional assumption about the variables that they are tracking. The algorithm assumes that the location variables are gaussian. This assumption simplifies the problem into one where the solution to where the other cars are can be computed exactly (and thus much faster).

Localization can be represented as the following iteration of sense, move and initial belief.

***“It first sets an initial belief senses with the measurements and then moves.”***



**3.3 Localisation Code: // Algorithm Replace**

**measurement=['r','r']**

**motion=[1,1]**

**def sense(p,n,z):**

**phit=0.8**

**pmiss=0.2**

**world=['g','r','r','g','g']**

**for i in range(n):**

**if(world[i]==z):**

**p[i]=p[i]\*phit**

**else:**

**p[i]=p[i]\*pmiss**

**s=sum(p)**

**for i in range(n):**

**p[i]/=s**

**return p**

**def move(p,u):**

**q = []**

**pexact=0.8**

**punder=0.1**

**pover=0.1**

**for i in range(len(p)):**

**s=pexact\*p[(i-u)%len(p)]**

**s=s+punder\*p[(i-u+1)%len(p)]**

**s=s+pover\*p[(i-u-1)%len(p)]**

**q.append(s)**

**return q**

**def func():**

**p=[]**

**n = input()**

**for i in range(n):**

**p.append(1.0/n)**

**print(p)**

**for i in range(len(measurement)):**

**p=sense(p,n,measurement[i])**

**print p**

**p=move(p,motion[i])**

**print p**

**#print p;**

**#return p**

**func()**

Chapter 4: Kalman Filter

**4.1 Introduction**

A Kalman filter is an optimal estimator - i.e. infers parameters of interest from indirect, inaccurate and uncertain observations. It is recursive so that new measurements can be processed as they arrive. (cf batch processing where all data must be present).

Optimization meaning: If all noise is Gaussian, the Kalman filter minimizes the mean square error of the estimated parameters.

**What if the noise is NOT Gaussian?**

Given only the mean and standard deviation of noise, the Kalman filter is the best linear estimator.

Non-linear estimators may be better.

**Why is Kalman Filtering so popular?**

* Good results in practice due to optimality and structure.
* Convenient form for online real time processing. ·
* Easy to formulate and implement given a basic understanding.
* Measurement equations need not be inverted.

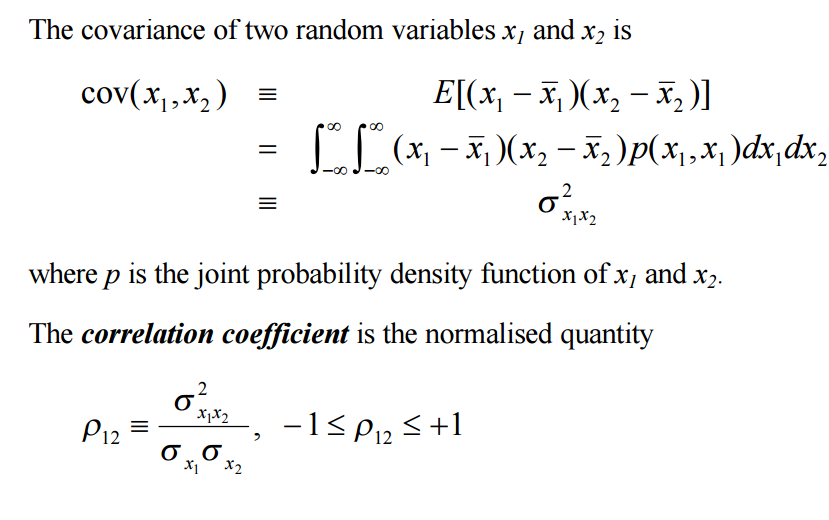
**Why use the word “Filter”?**

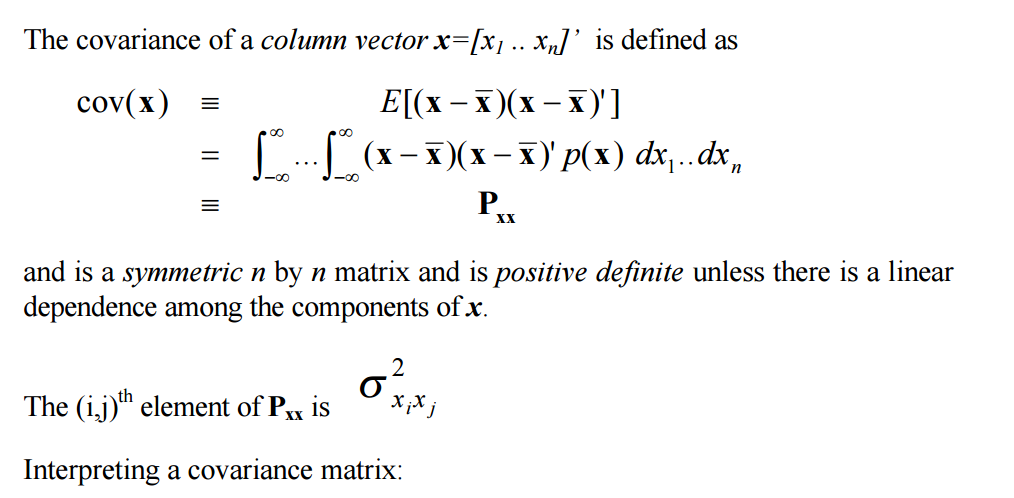
The process of finding the “best estimate” from noisy data amounts to “filtering out” the noise.

However a Kalman filter also doesn’t just clean up the data measurements, but also projects these measurements onto the state estimate.

**Co Variance:**

Co variance is determined as:





diagonal elements are the variances, off-diagonal encode correlations.

**Diagonalising a Covariance Matrix :**

cov(x) is symmetric => can be diagonalised using an orthonormal basis. By changing coordinates (pure rotation) to these unity orthogonal vectors we achieve decoupling of error contributions. The basis vectors are the eigenvectors and form the axes of error ellipses. The lengths of the axes are the square root of the eigenvalues and correspond to standard deviations of the independent noise contribution in the direction of the eigenvector.

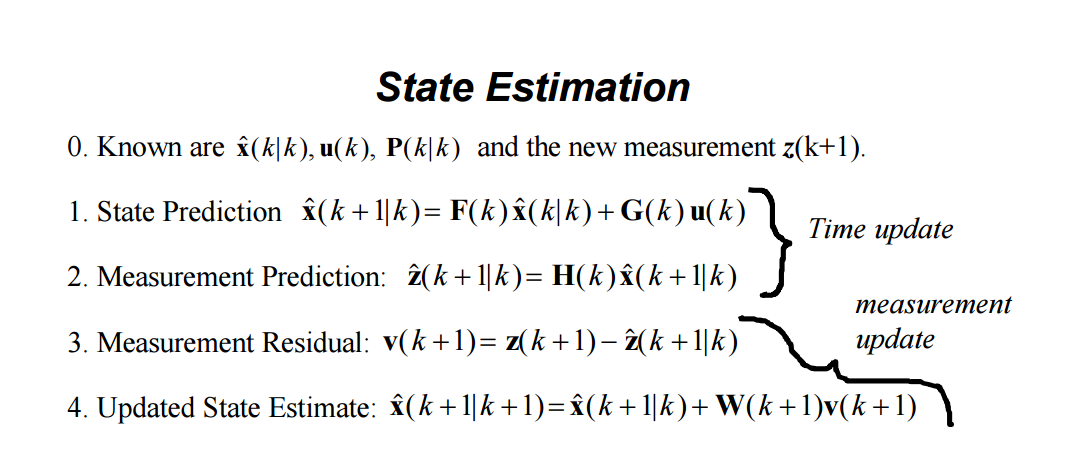
**KALMAN FILTER DEFINITION**

We require discrete time linear dynamic system description by vector difference equation with additive white noise that models unpredictable disturbances.

STATE DEFINITION -

The state of a deterministic dynamic system is the smallest vector that summarises the past of the system in full. Knowledge of the state allows theoretically prediction of the future (and prior) dynamics and outputs of the deterministic system in the absence of noise.

**4.2 Kalman Filter State Equations**



**4.3 Code:**

import numpy as NP

measurement=[1.,4.,7]

x=NP.matrix("0.; 0.")

p=NP.matrix("1000.0 0.0 ; 0.0 1000.0")

u=NP.matrix("0.; 0.")

F=NP.matrix("1. 1. ; 0. 1.")

H=NP.matrix("1. 0.")

r=NP.matrix("1.")

I=NP.matrix("1. 0. ; 0. 1.")

def func1():

filter(x,p)

def filter(x,p):

for i in range(len(measurement)):

#measurement update

Z=NP.matrix(measurement[i])

y=Z-H\*x

S=H\*p\*(H.T)+r

K=p\*(H.T)\*(S.I)

x=x+(K\*y)

p=(I-(K\*H))\*p

print ' measurement update '

print 'x= '

print x

print 'p= '

print p

#prediction update

x=(F\*x)+u

p=F\*p\*(F.T)

print ' motion update '

print 'x= '

print x

print 'p= '

print p

func1()

Chapter 5: Particle Filter

**5.1 Introduction**

In recent years, particle filters have solved several hard perceptual problems in robotics. Early successes of particle filters were limited to low-dimensional estimation problems, such as the problem of robot localization in environments with known maps.

More recently, researchers have begun exploiting structural properties of robotic domains that have led to successful particle filter applications in spaces with as many as 100,000 dimensions. The fact that every model—no mater how detailed—fails to capture the full complexity of even the most simple robotic environments has lead to specific tricks and techniques essential for the success of particle filters in robotic domains. This section covers some of these recent innovations, and provides pointers to in-depth articles on the use of particle filters in robotics.

Particle filters address the more general case of (nearly) unconstrained Markov chains. The basic idea is to approximate the posterior of a set of sample states x belonging to Xi, or particles. Here each is a concrete state sample of index i , where i’s range is the size of the particle filter.

The most basic version of particle filters is given by the following algorithm.

☛ Initialization: At time t=0, draw M particles according to

p(x0).

Call this set of particles X0

☛ Recursion: At time t>0 , generate a new particle for each already existing particle by drawing from the actuation model . Call the resulting set XT

Subsequently, draw M particles from Xi , so that each particle in X is drawn (with replacement) with a probability proportional to p( zi | xi ) .

Call the resulting set of particles Xt.

Particle filters are attractive to robot cists for more than one reason. First and foremost, they can be applied to almost any probabilistic robot model that can be formulated as a Markov chain. Furthermore, particle filters are anytime such that they do not require a fixed computation time; instead, their accuracy increases with the available computational resources. This makes them attractive to roboticists, who often face hard real-time constraints that have to be met using hard-to-control computer hardware.

Finally, they are relatively easy to implement. The implementer does not have to linearize non-linear models, and worry about closed-form solutions of the conditional statements of different form , as would be the case in Kalman filters, for example. The main criticism of particle filter has been that in general, populating a d-dimensional space requires exponentially many particles in d.

Most successful applications have therefore been confined to low-dimensional state spaces. The utilization of structure (e.g., conditional independences), present in many robotics problems, has only recently led to applications in higher dimensional spaces.

**PARTICLE FILTERS IN LOW DIMENSIONAL SPACES**

In robotics, the ‘classical’ successful example of particle filters is mobile robot localization. Mobile robot localization addresses the problem of estimation of a mobile robot’s pose relative to a given map from sensor measurements and controls. The pose is typically specified by a two-dimensional Cartesian coordinate and the robot’s rotational heading direction.

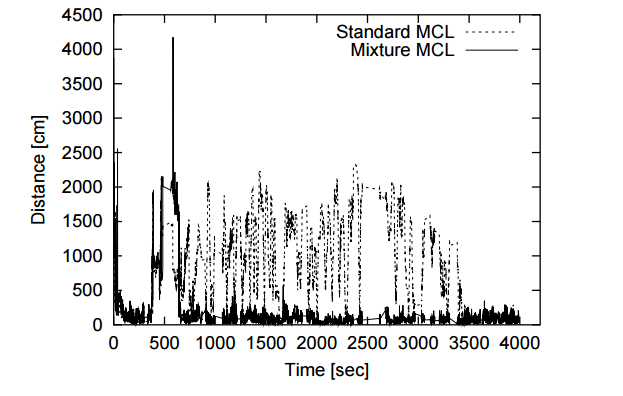
The problem is known as position tracking if the error can be guaranteed to be small at all times. More general is the global localization problem, which is the problem of localizing a robot under global uncertainty. The most difficult variant of the localization, however, is the kidnapped robot problem, in which a well-localized robot is tele-ported to some other location without being told. This problem was reported, for example, in the context of the Robocup soccer competition, in which judges picked up robots at random occasions and placed them somewhere else. Other localization problems involve multiple robots that can observe each other. 

Figure illustrates particle filters in the context of global localization of a robot in a known environment. Shown there is a progression of three situations, in which a number of particles approximate the posterior (1) at different stages of robot operation. Each particle is a sample of a three dimensional pose variable, comprising the robot’s Cartesian coordinates and its orientation relative to the map. The progression of snapshots in Figure 1 illustrate the development of the particle filter approximation over time, from global uncertainty to a well-localized robot.

In the context of localization, particle filters are commonly known as Monte Carlo localization (MCL). MCL’s original development was motivated by the condensation algorithm, a particle filter that enjoyed great popularity in computer vision applications. In most variants of the mobile localization problem, particle filters have been consistently found to outperform alternative techniques, including parametric probabilistic techniques such as the Kalman filter and more traditional techniques.

**PARTICLE FILTERS IN HIGH DIMENSIONAL SPACES**

An often criticized limitation of plain particle filters is their poor performance in higher dimensional spaces. This is because the number of particles needed to populate a state space scales exponentially with the dimension of the state space, not unlike the scaling limitations of vanilla HMMs. However, many problems in robotics possess structure that can be exploited to develop more efficient particle filters. One such problem is the simultaneous localization and mapping problem, or SLAM [8, 27, 36, 45]; see [48] for an overview. SLAM addresses the problem of building a map of the environment with a moving robot. The SLAM problem is challenging because errors in the robot’s localization induce systematic errors in the localization of environmental features in the map. The absence of an initial map in the SLAM problem makes it impossible to localize the robot during mapping using algorithms like MCL. Furthermore, the robot faces a challenging data association problem of determining whether two environment features, observed at different point in time, correspond to the same physical feature in the environment. To make matters worse, the space of all maps often comprises hundreds of thousand of dimensions. In the beginning of mapping the size of the state space is usually unknown, so SLAM algorithms have to estimate the dimensionality of the problem as well. On top of all this, most applications of SLAM require real-time processing.

**5.2 Code:**

from math import\*

import random

landmarks = [[20.0, 20.0], [80.0, 80.0], [20.0, 80.0], [80.0, 20.0]]

world\_size = 100.0

class robot:

# --------

# init:

# creates robot and initializes location/orientation

#

def \_\_init\_\_(self):

self.x = random.random() \* world\_size # initial x position

self.y = random.random() \* world\_size # initial y position

self.orientation = random.random() \* 2.0 \* pi # initial orientation

self.forward\_noise = 0.0

self.turn\_noise = 0.0

self.sense\_noise = 0.0

def \_\_repr\_\_(self):

return '[x=%.6s y=%.6s orient=%.6s]' % (str(self.x), str(self.y), str(self.orientation))

# --------

# set:

# sets a robot coordinate

#

def set(self, new\_x, new\_y, new\_orientation):

if new\_orientation < 0 or new\_orientation >= 2 \* pi:

raise ValueError, 'Orientation must be in [0..2pi]'

self.x = float(new\_x)

self.y = float(new\_y)

self.orientation = float(new\_orientation)

# --------

# set\_noise:

# sets the noise parameters

#

def set\_noise(self, new\_f\_noise, new\_t\_noise, new\_s\_noise):

# makes it possible to change the noise parameters

self.forward\_noise = float(new\_f\_noise)

self.turn\_noise = float(new\_t\_noise)

self.sense\_noise = float(new\_s\_noise)

# --------

# move:

# move along a section of a circular path according to motion

#

def move(self,turn,forward):

orientation=self.orientation+float(turn)+random.gauss(0.0,self.turn\_noise)

orientation%=2\*pi

dist=float(forward)+random.gauss(0.0,self.forward\_noise)

x=self.x+cos(orientation)\*dist

y=self.y+sin(orientation)\*dist

x%=world\_size

y%=world\_size

# set particle

res=robot()

res.set(x,y,orientation)

res.set\_noise(self.forward\_noise,self.turn\_noise,self.sense\_noise)

return res

def sense(self):

Z=[]

for i in range(len(landmarks)):

dist=sqrt((self.x-landmarks[i][0])\*\*2+(self.y-landmarks[i][1])\*\*2)

dist+=random.gauss(0.0,self.sense\_noise)

Z.append(dist)

return Z

def Gaussian(self,mu,sigma,x):

return (1/sqrt(2\*pi\*sigma\*\*2))\*exp(-1/2.0\*(x-mu)\*\*2/sigma\*\*2)

def measurement\_prob(self,measurement):

#determine how likely a measurement should be

prob=1.0

for i in range(len(landmarks)):

dist=sqrt((self.x-landmarks[i][0])\*\*2+(self.y-landmarks[i][1])\*\*2)

prob\*=self.Gaussian(dist,self.sense\_noise,measurement[i])

return prob

def eval(r,p): #determine the error

sum=0.0

for i in range(len(p)):

dx=min(abs(r.x-p[i].x),(world\_size-max(r.x,p[i].x)+min(r.x,p[i].x)))

dy=min(abs(r.y-p[i].y),(world\_size-max(r.y,p[i].y)+min(r.y,p[i].y)))

err=sqrt(dx\*dx+dy\*dy)

sum+=err

sum/=(len(p))

return sum

def func():

myrobot=robot()

#Initial particles

N=1000

T=100

p=[]

for i in range(N):

x=robot()

x.set\_noise(0.05,0.05,5.0)

p.append(x)

print eval(myrobot,p)

# Particle filtering after T steps : Both position and orientation matter

for t in range(T):

myrobot=myrobot.move(0.1,5.0)

Z=myrobot.sense()

p2=[]

for i in range(N):

x=p[i].move(0.1,5.0)

p2.append(x)

p=p2

w=[]

for i in range(N):

w.append(p[i].measurement\_prob(Z))

# Normalisation of intensity weights

su=sum(w)

for i in range(N):

w[i]/=su

# Resampling of particles : choosing particle for next state

p3=[]

index=int(random.random()\*N)

beta=0.0

mw=max(w)

for i in range(N):

beta+=(random.random()\*2\*mw)

while w[index]<beta :

beta-=w[index]

index=(index+1)%N

p3.append(p[index])

p=p3

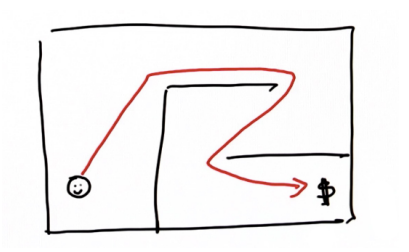
print eval(myrobot,p)

func()

Chapter 6: Motion Planning

**6.1 Introduction**

The fundamental problem in motion planning is that a robot might live in a world that looks like the one pictured below and will want to find a goal like $. The robot has to have a plan to get to its goal, as indicated by the red line route.



This same problem occurs for a self-driving car that lives in a city and has to find its way to its target location by navigating through a network of streets and along the highway.

**Given**:

* Map Starting Location
* Goal Location Cost (time it takes to drive a certain route)
* **Goal**: Find the minimum cost path.

**Computing Cost (Compute Cost) :**

Suppose you live in a discrete world that is split up into grid cells and your initial location is facing forward, while your goal location is facing to the left.

Assume that for each time step you can make one of two action -- either move forward or turn the vehicle. Each move costs exactly one unit.

**Writing a Search Program (First Search Program)**

Now, the question is, can you write a program that computes the shortest path from the start to the goal? The first step towards writing this program is to name the grid cells. Across the top, name the columns zero to five, and along the side you can name the rows zero to four. Think about each grid cell as a data point, called a node and the goal point is called the goal node.

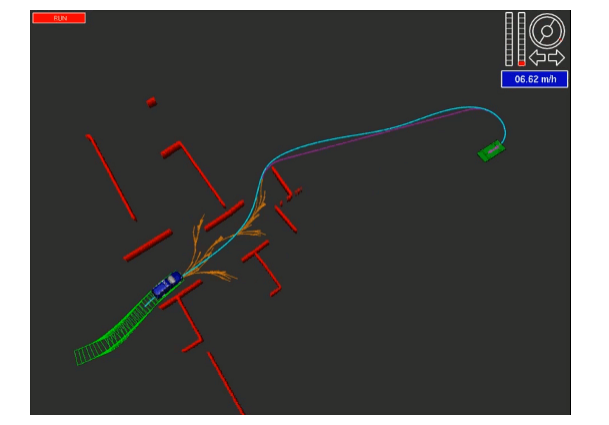
1. From here you will make a list of nodes that you will be investigating further -- that you will be expanding. Call this list open. The initial state of this list is [0,0]. You may want to check off the nodes that you have already picked.
2. Now, you can test whether this node is your final goal node -- which, just by looking at the grid you can tell that it obviously is not. So, what you want to do next is to expand this node by taking it off the open list and looking at its successors
3. Then, you can check those cells off on your grid. Most importantly, remember to note how many expansions it takes to get to the goal -- this is called the g-value. By the end of your planning, the g-value will be the length of the path
4. The process of expanding the cells, starting with the one with the lowest g-value, should yield the same result you found as the number of moves it takes to get from the start to the goal.

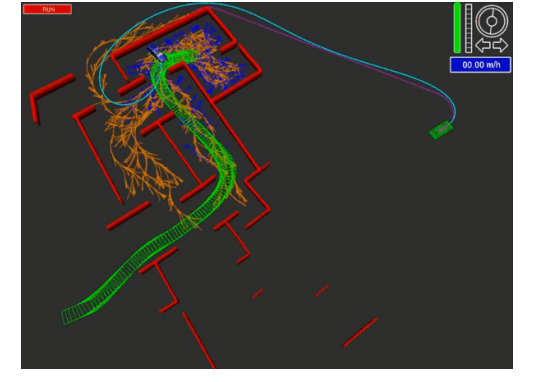
**6.2 A\* search algorithm (A Star) :**

The A\* (A-star) algorithm can be used for our path finding problem. It is a variant of the search algorithm that we implemented, that is more efficient because you don’t need to expand every node. The A\* algorithm was first described by Peter Hart, Nils Nilsson and Bertram Raphael in 1968. If you understand the mechanism for searching by gradually expanding the nodes in the open list, A\* is almost the same thing, but not quite.

A\* uses a heuristic function, which is a function that has to be set up. If it’s all zeroes, A\* resorts back to the search algorithm already implemented. If we call the heuristic function h, then each cell results in a value.

**Examples of A\* in the real world :**





**6.3 Code:**

from math import\*

import random

# Motion Planning Technique

def search(grid,init,goal):

# open list items are of type [g,x,y]

closed=[[0 for x in range(len(grid[0]))] for y in range(len(grid))]

expand=[[-1 for x in range(len(grid[0]))] for y in range(len(grid))]

policy=[[' ' for x in range(len(grid[0]))] for y in range(len(grid))]

action=[[-1 for x in range(len(grid[0]))] for y in range(len(grid))]

closed[init[0]][init[1]]=1

g=0

x=init[0]

y=init[1]

cost=1

open=[[g,x,y]]

delta=[[-1,0],[0,-1],[1,0],[0,1]]

delta\_name=['^','<','v','>']

found=0

resign=0

count=0

while(found==0 and resign==0):

# no element in next element list

if len(open)==0:

resign=1

print 'fail'

else :

open.sort()

open.reverse()

next=open.pop() #list element with minimum g value

#print next

x=next[1]

y=next[2]

g=next[0]

expand[x][y]=count

count+=1

if x==goal[0] and y==goal[1]:

found=1

print next

else:

for i in range(len(delta)): # 4 directions

x2=x+delta[i][0]

y2=y+delta[i][1]

if x2>=0 and x2<len(grid) and y2>=0 and y2<len(grid[0]): # grid range

if closed[x2][y2]==0 and grid[x2][y2]==0:

g2=g+cost

open.append([g2,x2,y2])

closed[x2][y2]=1

action[x2][y2]=i

# expansion grid

for i in range(len(grid)):

print expand[i]

#print path

#---------------------------------------------------------------------------------------------

x=goal[0]

y=goal[1]

policy[x][y]='\*'

while x!=init[0] or y!=init[1]:

x2=x-delta[action[x][y]][0]

y2=y-delta[action[x][y]][1]

policy[x2][y2]=delta\_name[action[x][y]]

x=x2

y=y2

for i in range(len(grid)):

print policy[i]

#------------------------------------------------------------------------------------

def A\_star\_search(grid,init,goal):

# open list items are of type [g,x,y]

heuristic=[[9,8,7,6,5,4],

[8,7,9,5,4,3],

[7,6,5,4,3,2],

[6,5,4,3,2,1],

[5,4,3,2,1,0]]

closed=[[0 for x in range(len(grid[0]))] for y in range(len(grid))]

expand=[[-1 for x in range(len(grid[0]))] for y in range(len(grid))]

policy=[[' ' for x in range(len(grid[0]))] for y in range(len(grid))]

action=[[-1 for x in range(len(grid[0]))] for y in range(len(grid))]

closed[init[0]][init[1]]=1

g=0

x=init[0]

y=init[1]

cost=1

f=g+heuristic[x][y]

open=[[f,g,x,y]]

delta=[[-1,0],[0,-1],[1,0],[0,1]]

delta\_name=['^','<','v','>']

found=0

resign=0

count=0

while(found==0 and resign==0):

# no element in next element list

if len(open)==0:

resign=1

print 'fail'

else :

open.sort()

open.reverse()

next=open.pop() #list element with minimum g value

#print next

x=next[2]

y=next[3]

g=next[1]

expand[x][y]=count

count+=1

if x==goal[0] and y==goal[1]:

found=1

print next

else:

for i in range(len(delta)): # 4 directions

x2=x+delta[i][0]

y2=y+delta[i][1]

if x2>=0 and x2<len(grid) and y2>=0 and y2<len(grid[0]): # grid range

if closed[x2][y2]==0 and grid[x2][y2]==0:

g2=g+cost

open.append([g2+heuristic[x2][y2],g2,x2,y2])

closed[x2][y2]=1

action[x2][y2]=i

# expansion grid

for i in range(len(grid)):

print expand[i]

#print path

#---------------------------------------------------------------------------------------------

x=goal[0]

y=goal[1]

policy[x][y]='\*'

while x!=init[0] or y!=init[1]:

x2=x-delta[action[x][y]][0]

y2=y-delta[action[x][y]][1]

policy[x2][y2]=delta\_name[action[x][y]]

x=x2

y=y2

for i in range(len(grid)):

print policy[i]

#------------------------------------------------------------------------------------

def Dynamic\_progsearch(grid,init,goal):

value=[[99 for x in range(len(grid[0]))] for y in range(len(grid))]

policy=[[' ' for x in range(len(grid[0]))] for y in range(len(grid))]

cost\_step=1

change=1

delta=[[-1,0],[0,-1],[1,0],[0,1]]

delta\_name=['^','<','v','>']

# calcluate optimal value from each location

while change==1:

change=0

for x in range(len(grid)):

for y in range(len(grid[0])):

if goal[0]==x and goal[1]==y:

if value[x][y]>0:

value[x][y]=0

change=1

elif grid[x][y]==0:

for a in range(len(delta)):

x2=x+delta[a][0]

y2=y+delta[a][1]

if x2>=0 and x2<len(grid) and y2>=0 and y2<len(grid[0]) and grid[x2][y2]==0:

v2=value[x2][y2]+cost\_step

if v2<value[x][y]:

change=1

value[x][y]=v2

for i in range(len(grid)):

print value[i]

# calculate policy matrix in case car reaches any position on grid

for x in range(len(grid)):

for y in range(len(grid[0])):

if x==goal[0] and y==goal[1]:

policy[x][y]='\*'

elif grid[x][y]==0:

dir=' '

val=99

for i in range(len(delta)):

x2=x+delta[i][0]

y2=y+delta[i][1]

if x2>=0 and x2<len(grid) and y2>=0 and y2<len(grid[0]) and grid[x2][y2]==0:

if value[x2][y2]<val:

val=value[x2][y2]

dir=delta\_name[i]

policy[x][y]=dir

for i in range(len(grid)):

print policy[i]

#-----------------------------------------------------------------------------------------------------

def three\_D\_Dynamic\_prog():

grid=[[1,1,1,0,0,0],

[1,1,1,0,1,0],

[0,0,0,0,0,0],

[1,1,1,0,1,1],

[1,1,1,0,1,1]]

value=[[[999 for x in range(len(grid[0]))] for y in range(len(grid))],

[[999 for x in range(len(grid[0]))] for y in range(len(grid))],

[[999 for x in range(len(grid[0]))] for y in range(len(grid))],

[[999 for x in range(len(grid[0]))] for y in range(len(grid))]]

policy=[[[' ' for x in range(len(grid[0]))] for y in range(len(grid))],

[[' ' for x in range(len(grid[0]))] for y in range(len(grid))],

[[' ' for x in range(len(grid[0]))] for y in range(len(grid))],

[[' ' for x in range(len(grid[0]))] for y in range(len(grid))]]

policy2D=[[' ' for x in range(len(grid[0]))] for y in range(len(grid))]

goal=[4,5]

init=[0,0,2]

cost=[1,1,1]

action=[-1,0,1]

action\_name=['R','#','L']

forward=[[-1,0],[0,-1],[1,0],[0,1]]

change=1

while change==1:

change=0

for x in range(len(grid)):

for y in range(len(grid[0])):

for orientation in range(4):

if goal[0]==x and goal[1]==y:

if value[orientation][x][y]>0:

value[orientation][x][y]=0

change=1

policy[orientation][x][y]='\*'

elif grid[x][y]==0:

for i in range(3):

o2=(orientation+action[i])%4

x2=x+forward[o2][0]

y2=y+forward[o2][1]

if x2>=0 and x2<len(grid) and y2>=0 and y2<len(grid[0]) and grid[x2][y2]==0:

v2=value[o2][x2][y2]+cost[i]

if v2<value[orientation][x][y]:

value[orientation][x][y]=v2

policy[orientation][x][y]=action\_name[i]

change=1

#---------policy 2-D array

x=init[0]

y=init[1]

orientation=init[2]

policy2D[x][y]=policy[orientation][x][y]

while policy[orientation][x][y]!='\*':

if policy[orientation][x][y]=='#':

o2=orientation

elif policy[orientation][x][y]=='R':

o2=(orientation-1)%4

elif policy[orientation][x][y]=='L':

o2=(orientation+1)%4

x=x+forward[o2][0]

y=y+forward[o2][1]

orientation=o2

policy2D[x][y]=policy[orientation][x][y]

for i in range(len(policy2D)):

print policy2D[i]

#------------------------------------------------------------------------------------------------

def func():

grid=[[1,1,1,0,0,0],

[1,1,1,0,1,0],

[0,0,0,0,0,0],

[1,1,1,0,1,1],

[1,1,1,0,1,1]]

init=[0,0]

goal=[len(grid)-1,len(grid[0])-1]

#search(grid,init,goal)

#A\_star\_search(grid,init,goal)

#Dynamic\_progsearch(grid,init,goal)

three\_D\_Dynamic\_prog()

func()

Chapter 7: PID Controller

**7.1 Introduction**

A **proportional–integral–derivative controller** (**PID controller**) is a [control loop](https://en.wikipedia.org/wiki/Control_loop) [feedback mechanism](https://en.wikipedia.org/wiki/Feedback_mechanism) ([controller](https://en.wikipedia.org/wiki/Controller_(control_theory))) commonly used in [industrial control systems](https://en.wikipedia.org/wiki/Industrial_control_system). A PID controller continuously calculates an *error value* as the difference between a desired [setpoint](https://en.wikipedia.org/wiki/Setpoint_(control_system)) and a measured [process variable](https://en.wikipedia.org/wiki/Process_variable). The controller attempts to minimize the error over time by adjustment of a *control variable*, such as the position of a [control valve](https://en.wikipedia.org/wiki/Control_valve), a [damper](https://en.wikipedia.org/wiki/Damper_(flow)), or the power supplied to a heating element, to a new value determined by a weighted sum:

u(t) = K_p e(t) + K_i \int_{0}^{t}e(\tau)d\tau + K_d \frac{de(t)}{dt}

WhereK_p, K_i, and K_d, all non-negative, denote the coefficients for the [proportional](https://en.wikipedia.org/wiki/Proportional_control), [integral](https://en.wikipedia.org/wiki/Integral), and [derivative](https://en.wikipedia.org/wiki/Derivative) terms, respectively (sometimes denoted *P,* *I,* and *D*). In this model,

* *P* accounts for present values of the error. For example, if the error is large and positive, the control output will also be large and positive.
* *I* accounts for past values of the error. For example, if the current output is not sufficiently strong, error will accumulate over time, and the controller will respond by applying a stronger action.
* *D* accounts for possible future values of the error, based on its current rate of change.[[1]](https://en.wikipedia.org/wiki/PID_controller#cite_note-1)

As a PID controller relies only on the measured process variable, not on knowledge of the underlying process, it is broadly applicable.[[2]](https://en.wikipedia.org/wiki/PID_controller#cite_note-ben93p48-2) By tuning the three parameters of the model, a PID controller can deal with specific process requirements. The response of the controller can be described in terms of its responsiveness to an error, the degree to which the system [overshoots](https://en.wikipedia.org/wiki/Overshoot_(signal)) a setpoint, and the degree of any system oscillation. The use of the PID algorithm does not guarantee [optimal control](https://en.wikipedia.org/wiki/Optimal_control) of the system or even its [stability](https://en.wikipedia.org/wiki/Nyquist_stability_criterion).

Some applications may require using only one or two terms to provide the appropriate system control. This is achieved by setting the other parameters to zero. A PID controller will be called a PI, PD, P or I controller in the absence of the respective control actions. PI controllers are fairly common, since derivative action is sensitive to measurement noise, whereas the absence of an integral term may prevent the system from reaching its target value.

**7.2 PID Control Theory**

The PID control scheme is named after its three correcting terms, whose sum constitutes the manipulated variable (MV). The proportional, integral, and derivative terms are summed to calculate the output of the PID controller. Defining u(t) as the controller output, the final form of the PID algorithm is:

\mathrm{u}(t)=\mathrm{MV}(t)=K_p{e(t)} + K_{i}\int_{0}^{t}{e(\tau)}\,{d\tau} + K_{d}\frac{de(t)}{dt}

Where,

K_p: Proportional gain, a tuning parameter

K_i: Integral gain, a tuning parameter

K_d: Derivative gain, a tuning parameter

e: Error  = SP - PV 

SP: Set Point

PV: Process Variable

t: Time or instantaneous time (the present)

: Variable of integration; takes on values from time 0 to the present t.



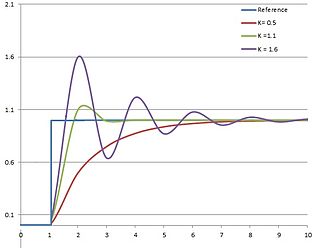
Equivalently, the transfer function in the [Laplace Domain](https://en.wikipedia.org/wiki/Laplace_Transform) of the PID controller is

L(s)=K_p + K_{i}/s + K_{d}s

where

s: complex number frequency

**Proportional term**

[](https://en.wikipedia.org/wiki/File:PID_varyingP.jpg)

Plot of PV vs time, for three values of Kp (Ki and Kdheld constant)

The proportional term produces an output value that is proportional to the current error value. The proportional response can be adjusted by multiplying the error by a constant *Kp*, called the proportional gain constant.

The proportional term is given by:

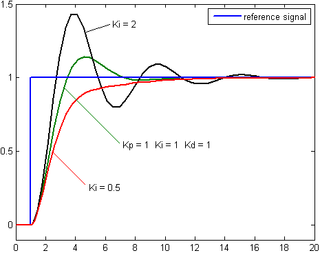
P_{\mathrm{out}}=K_p\,{e(t)}

A high proportional gain results in a large change in the output for a given change in the error. If the proportional gain is too high, the system can become unstable (see [the section on loop tuning](https://en.wikipedia.org/wiki/PID_controller#Loop_tuning)). In contrast, a small gain results in a small output response to a large input error, and a less responsive or less sensitive controller. If the proportional gain is too low, the control action may be too small when responding to system disturbances. Tuning theory and industrial practice indicate that the proportional term should contribute the bulk of the output change.

**Steady-state error**

Because a non-zero error is required to drive it, a proportional controller generally operates with a so-called *steady-state error*. Steady-state error (SSE) is proportional to the process gain and inversely proportional to proportional gain. SSE may be mitigated by adding a compensating [bias term](https://en.wikipedia.org/wiki/Biasing) to the setpoint or output, or corrected dynamically by adding an integral term.

**Integral term**

[](https://en.wikipedia.org/wiki/File:Change_with_Ki.png)

Plot of PV vs time, for three values of Ki (Kp and Kdheld constant)

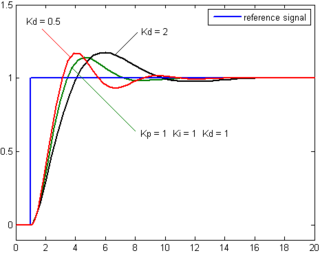
The contribution from the integral term is proportional to both the magnitude of the error and the duration of the error. The [integral](https://en.wikipedia.org/wiki/Integral) in a PID controller is the sum of the instantaneous error over time and gives the accumulated offset that should have been corrected previously. The accumulated error is then multiplied by the integral gain (K_i) and added to the controller output.

The integral term is given by:

I_{\mathrm{out}}=K_{i}\int_{0}^{t}{e(\tau)}\,{d\tau}

The integral term accelerates the movement of the process towards setpoint and eliminates the residual steady-state error that occurs with a pure proportional controller. However, since the integral term responds to accumulated errors from the past, it can cause the present value to [overshoot](https://en.wikipedia.org/wiki/Overshoot_(signal)) the setpoint value (see [the section on loop tuning](https://en.wikipedia.org/wiki/PID_controller#Loop_tuning)).

**Derivative term**

[](https://en.wikipedia.org/wiki/File:Change_with_Kd.png)

Plot of PV vs time, for three values of Kd (Kp and Kiheld constant)

The [derivative](https://en.wikipedia.org/wiki/Derivative) of the process error is calculated by determining the slope of the error over time and multiplying this rate of change by the derivative gain *Kd*. The magnitude of the contribution of the derivative term to the overall control action is termed the derivative gain, *Kd*.

The derivative term is given by:

D_{\mathrm{out}}=K_d\frac{de(t)}{dt}

Derivative action predicts system behavior and thus improves settling time and stability of the system. An ideal derivative is not [causal](https://en.wikipedia.org/wiki/Causal_system), so that implementations of PID controllers include an additional low pass filtering for the derivative term, to limit the high frequency gain and noise.[[14]](https://en.wikipedia.org/wiki/PID_controller#cite_note-IEEE_PID_05-15) Derivative action is seldom used in practice though - by one estimate in only 25% of deployed controllers - because of its variable impact on system stability in real-world applications.

**Loop tuning**

*Tuning* a control loop is the adjustment of its control parameters (proportional band/gain, integral gain/reset, derivative gain/rate) to the optimum values for the desired control response. Stability (no unbounded oscillation) is a basic requirement, but beyond that, different systems have different behavior, different applications have different requirements, and requirements may conflict with one another.

PID tuning is a difficult problem, even though there are only three parameters and in principle is simple to describe, because it must satisfy complex criteria within the [limitations of PID control](https://en.wikipedia.org/wiki/PID_controller#Limitations_of_PID_control). There are accordingly various methods for loop tuning, and more sophisticated techniques are the subject of patents; this section describes some traditional manual methods for loop tuning.

Designing and tuning a PID controller appears to be conceptually intuitive, but can be hard in practice, if multiple (and often conflicting) objectives such as short transient and high stability are to be achieved. PID controllers often provide acceptable control using default tunings, but performance can generally be improved by careful tuning, and performance may be unacceptable with poor tuning. Usually, initial designs need to be adjusted repeatedly through computer simulations until the closed-loop system performs or compromises as desired.

Some processes have a degree of [nonlinearity](https://en.wikipedia.org/wiki/Nonlinear_system) and so parameters that work well at full-load conditions don't work when the process is starting up from no-load; this can be corrected by [gain scheduling](https://en.wikipedia.org/wiki/Gain_scheduling) (using different parameters in different operating regions).

**Stability**

If the PID controller parameters (the gains of the proportional, integral and derivative terms) are chosen incorrectly, the controlled process input can be unstable, i.e., its output [diverges](https://en.wikipedia.org/wiki/Divergence_(computer_science)), with or without [oscillation](https://en.wikipedia.org/wiki/Oscillation), and is limited only by saturation or mechanical breakage. Instability is caused by *excess* gain, particularly in the presence of significant lag.

Generally, stabilization of response is required and the process must not oscillate for any combination of process conditions and setpoints, though sometimes [marginal stability](https://en.wikipedia.org/wiki/Marginal_stability) (bounded oscillation) is acceptable or desired.

Mathematically, the origins of instability can be seen in the Laplace domain. The total loop transfer function is:

H(s)=\frac{K(s)G(s)}{1+K(s)G(s)}

Where,

K(s): PID transfer function

: Plant transfer function



The system is called unstable where the closed loop transfer function diverges for some s. This happens for situations where K(s)G(s)=-1. Typically, this happens when |K(s)G(s)|=1 with a 180 degree phase shift. Stability is guaranteed when K(s)G(s)<1 for frequencies that suffer high phase shifts. A more general formalism of this effect is known as the [Nyquist stability criterion](https://en.wikipedia.org/wiki/Nyquist_stability_criterion).

**Optimum behavior**

The optimum behavior on a process change or setpoint change varies depending on the application.

Two basic requirements are *regulation* (disturbance rejection – staying at a given setpoint) and *command tracking* (implementing setpoint changes) – these refer to how well the controlled variable tracks the desired value. Specific criteria for command tracking include [rise time](https://en.wikipedia.org/wiki/Rise_time) and [settling time](https://en.wikipedia.org/wiki/Settling_time). Some processes must not allow an overshoot of the process variable beyond the setpoint if, for example, this would be unsafe. Other processes must minimize the energy expended in reaching a new setpoint.

Chapter 8: Conclusion and Future Wok

The software implemented will be a key part in designing the self-driving car. The techniques used for particle tracking , motion planning and smoothing minimizes error rates to maximum extent and helps in designing a reliable Working self – driving car . Except for the software implementation, a lot goes into implementing the actual car itself .

Efforts can be made to further optimize the error rates in future so that such cars can be used in future in general traffic.

Chapter 9: References

**Journal :**

[1] <https://www.udacity.com/course/artificial-intelligence-for-robotics--cs373>

**Article:**

[2]<http://en.wikipedia.org/wiki/Autonomous_car>

**Research Paper**

[3]<http://robohub.org/how-do-self-driving-cars-work/>

**Article**

[4]<http://biorobotics.ri.cmu.edu/papers/sbp_papers/integrated3/kleeman_kalman_basics.pdf>

**Article:**

[5]<http://robots.stanford.edu/papers/thrun.pf-in-robotics-uai02.pdf>

**Article:**

[6]<http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>

**Article:**

[7]<http://en.wikipedia.org/wiki/PID_controller>

**Research Paper :**

[8] Sensor Fusion of Laser and Stereo Vision Camera , By Mrs.Daya Gupta and Sakshi Yadav

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.184.5692&rep=rep1&type=pdf>